Dynamo: Amazon’s Highly Available Key-value Store

Suresh Podeti

10 min read · Sep 22, 2023

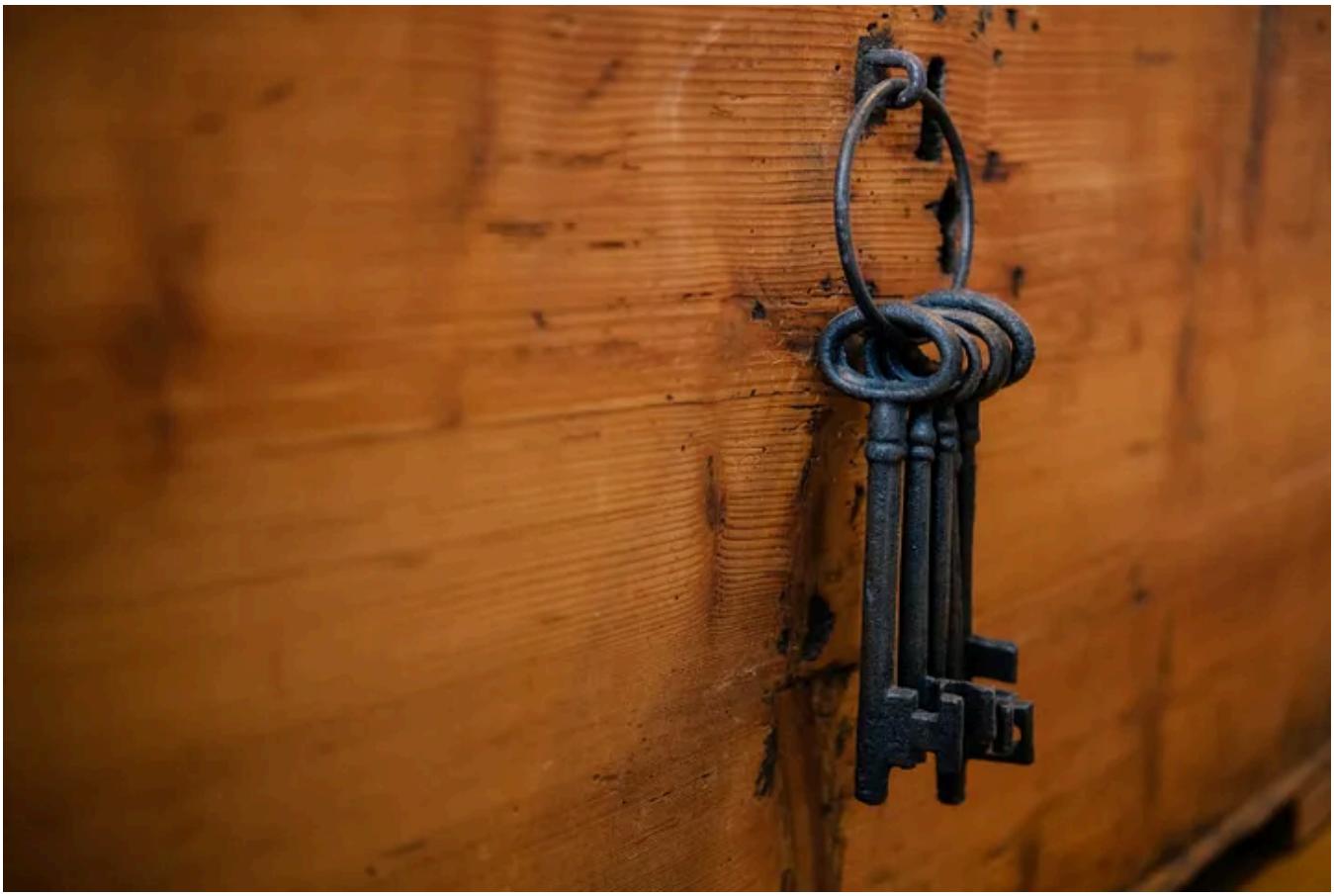

Photo by Silas Köhler on Unsplash

Introduction

This article presents the design and implementation of Dynamo, a highly available key-value storage system that some of amazon’s core services use to provide an “always-on” experience.

Key-value stores are **distributed hash tables** (DHTs). A **key** is generated by the **hash** function and should be **unique**. In a key-value store, a key binds to a specific value and doesn't assume anything about the structure of the value.

Dynamo uses **MD5** hashes on the key to generate a **128-bit identifier**. These identifiers help the system determine which server node will be responsible for this specific key

Requirements

Non-functional

- **Scalable**: Our system should be able to handle an enormous number of users of the key-value store.
- **Available**: We need to provide continuous service, so availability is very important.
- **Fault tolerance**: The key-value store should operate uninterrupted despite failures in servers or their components.

API design

The `get` function

The API call to get a value should look like this:

```
get(key)
```

The `put` function

The API call to put the value into the system should look like this:

```
put(key, value)
```

Summary

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability gurantee when some oof the replicas are not avaialble
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information

Fig 1.0: Summary of the list of techniques Dynamo uses and their respective advantages

Scalability

To ensure scalability for a large user base, it's essential to partition our data across multiple nodes within the system.

Traditional way to solve this is through the **modulus** operator (% mod). When request comes in, we assign a request ID, calculate its hash, and find the reminder by taking the modulus with the number of nodes available. Then reminder value is the node number, and we send the request to the node to process it.

```
node_number = h(requestId) % no.of nodes
```

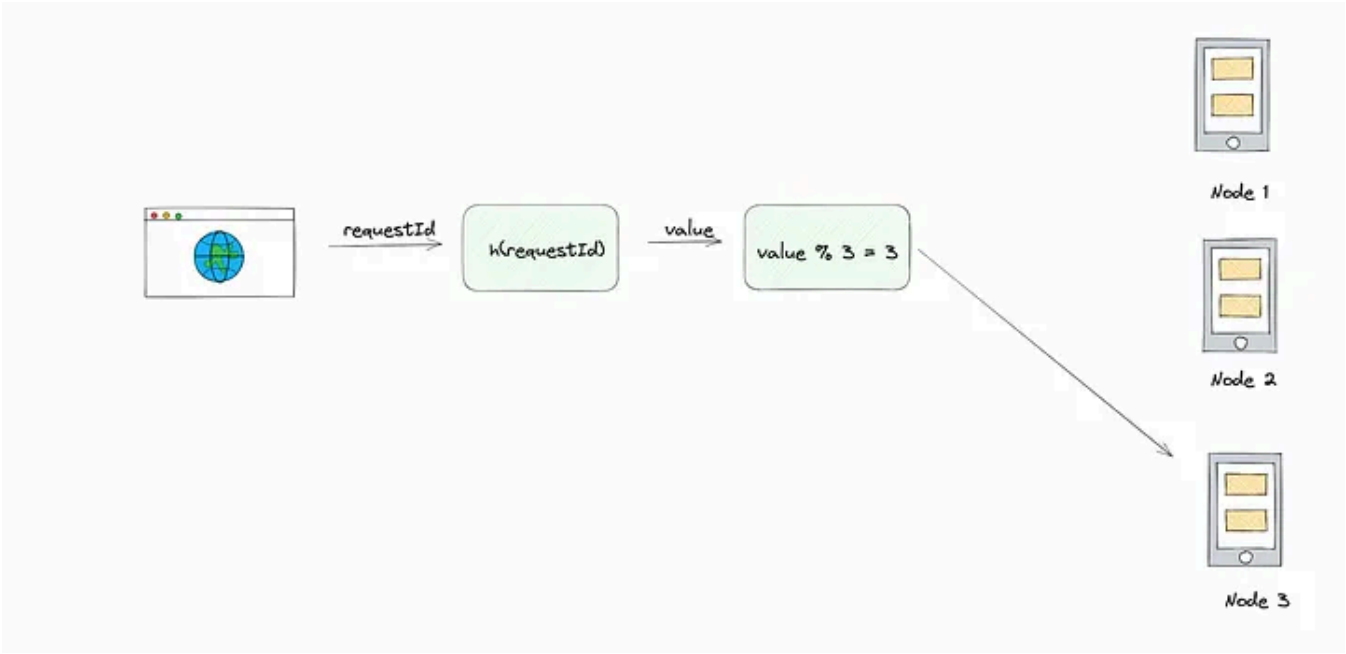

Fig 2.0: We get the hash of the requestID and take modulus with the number of nodes to find the node that should process the request

We want to add and remove nodes with minimal change in our infrastructure. But in this method, when we add or remove a node, we need to move a lot of keys. This is inefficient.

Consistent hashing

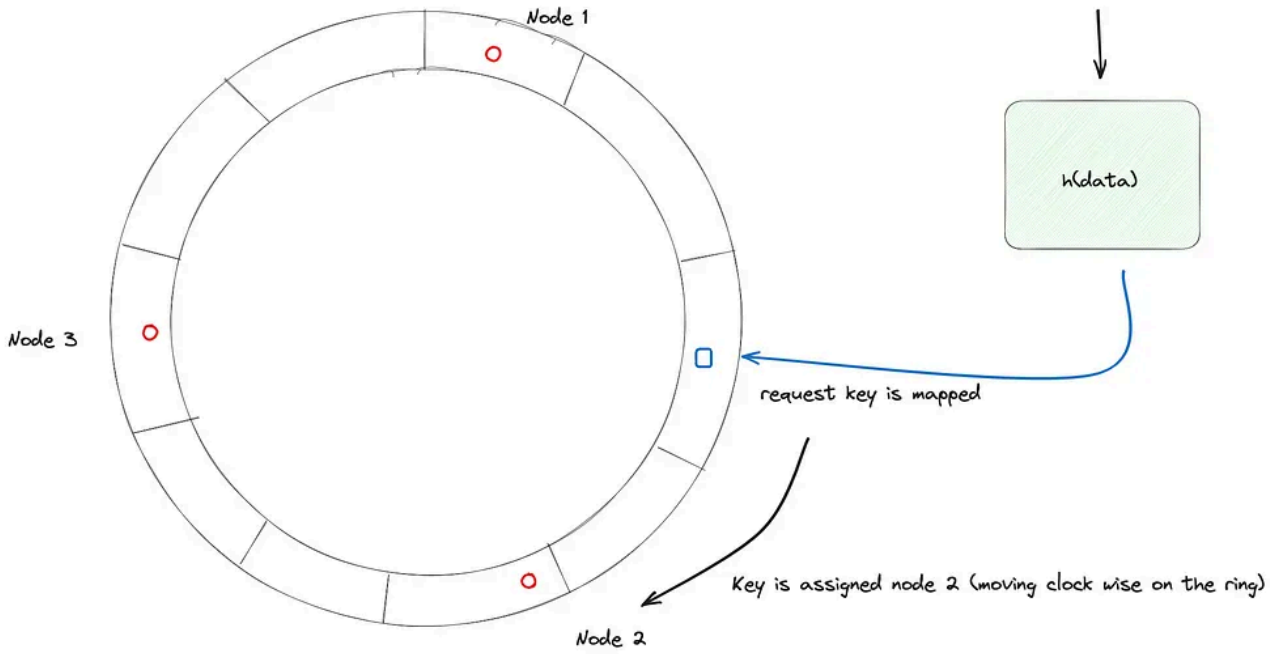

Fig 3.0: Consider we have a conceptual ring of hashes from 0 to n-1, where n is the total number of hash values in the ring

In consistent hashing, we consider that we have a **conceptual ring** of hashes from 0 to n−1, where n is the number of available hash values. We use each node's ID, calculate its hash, and map it to the ring. We apply the same process to requests. Each request is completed by the next node that it finds by moving in the clockwise direction in the ring.

Whenever a new node is added to the ring, the immediate next node is affected. It has to share its data with the newly added node while other nodes are unaffected. It's easy to scale since we're able to keep changes to our nodes minimal. This is because only a small portion of overall keys need to move.

However, the request load isn't equally divided in practice. Any server that handles a large chunk of data can become a bottleneck in a distributed system. That node will receive a disproportionately large share of data storage and retrieval requests, reducing the overall system performance. As a result, these are referred to as hotspots.

Use virtual nodes

We'll use virtual nodes to ensure a more evenly distributed load across the nodes. Instead of applying a single hash function, we'll apply multiple hash functions onto the same node.

For the request, we use only one hash function. This strategy provides approximate uniform distribution.

We've made the proposed design of key-value storage scalable. The next task is to make our system highly available.

Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. All involved storage areas are primary, and they replicate the data to stay updated. Both read and write are allowed on all nodes. Usually, it's inefficient and costly to replicate in all the nodes. **Each data item is replicated at N hosts**, where N is a parameter configured "*per-instance*". Each key K is assigned to a coordinator node, which is in charge of the replication of data items that fall within its range these keys at N-1 clockwise successor nodes in the ring along with storing it locally. This results in a system where each node is responsible for the region of the ring between it and its Nth predecessor.

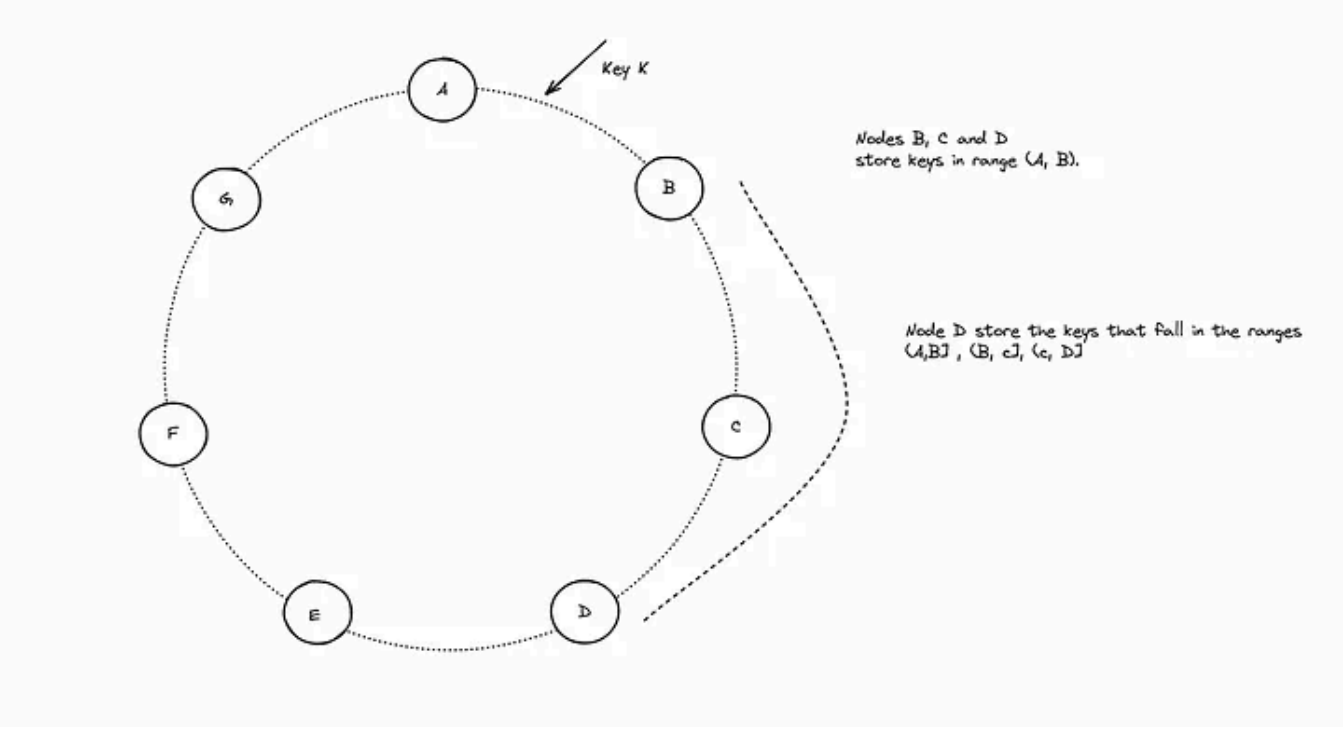

Fig 4.0: Replication of keys in dynamo ring

The list of nodes that is responsible for storing a particular key is called the *preference list.* It is possible that the first N successor positions for particular key may

be less than N distinct physical nodes. To address this, the preference list for a key is constructed by skipping the positions in the ring to ensure that the list contains only distinct physical nodes.

Data versioning

Dynamo provides **eventual consistency,** which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates. In case of node failures, other node in the preference list handles the writes. If most recent state of the data unavailable, and if we make change to an older version of the data, that change is still meaningful and should be preserved.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a **new and immutable** version of the data. It allows multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version . However, aversion branching may happen, in the presence of conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to collapse multiple branches of data evolution back into one.

Vector clock

Dynamo uses **vector clocks** in order to capture the **causality** between different versions of the same object.

A vector clock is effectively a list of counters on each node (node, counter). One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causual ordering, by examine their vector clocks. If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, the n the first is an ancestor of the seoncd and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

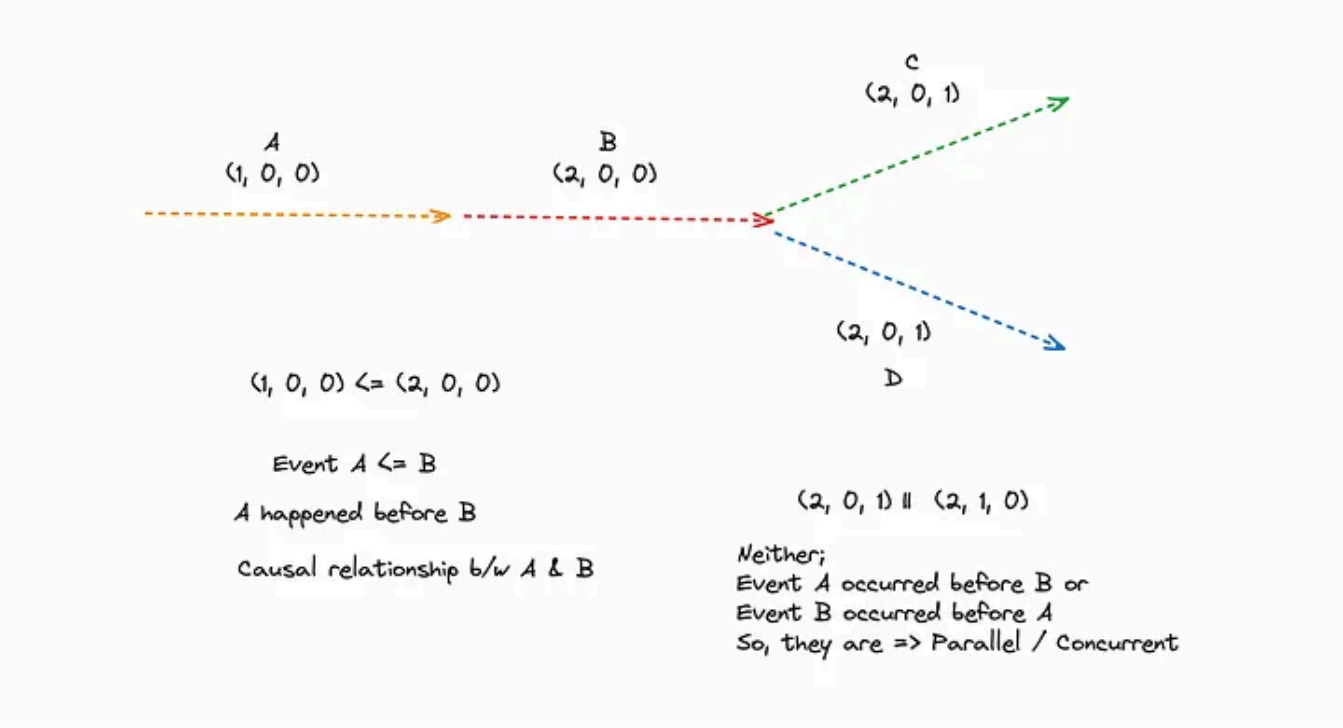

Fig 5.0: Vector clocks used to capture causality between events

In dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information.

Upon processing a read request, if Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects in the leaves with the corresponding version inforation in the context. An udpate using this context is considered to have reconciled the divergent versions and the brances are collapsed into a single new version.

Example

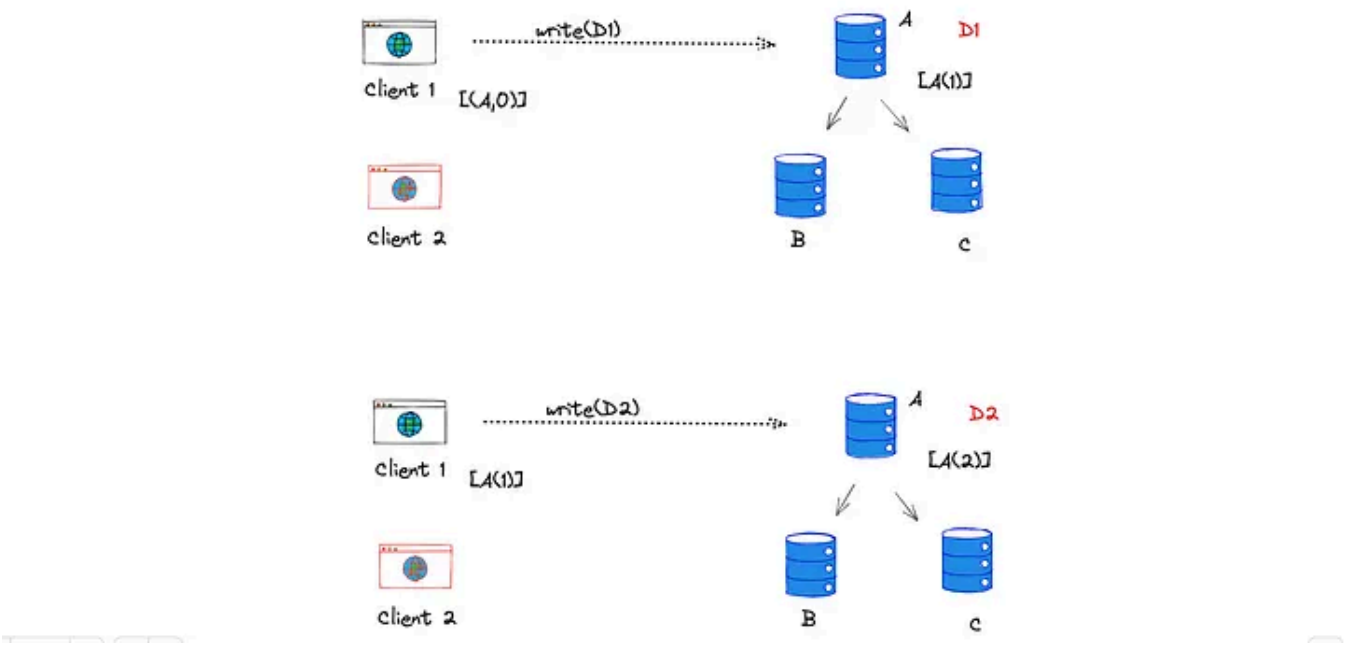

Fig 6.0: Client 1 writes two updates to same node A

Fig 7.0: Client 1 and Client 2 reads version [A(2)] at the same time update to different nodes (B and C respectively). Vector clocks helps to find they are concurrent updates eg: (2, 1, 0) and (2, 0, 1) are concurrent events there is no causal relation between them

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. Writes are usually handled by one of the top N nodes in the preference list. In case of network failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the **size of the vector clock to grow**. In these scenarios, it is desirable to limit the size of vector clock. To this, Dynamo employs the following **clock truncation** scheme:

Along with each node (node, counter) pair, dynamo **stores a timestamp** that indicates that last time the node updated the data item. When the number of (node,

counter) pairs in the vector clock reaches thread (say 10), the oldest pair is removed from clock. Demerit — Can lead to inefficiencies in reconciliation.

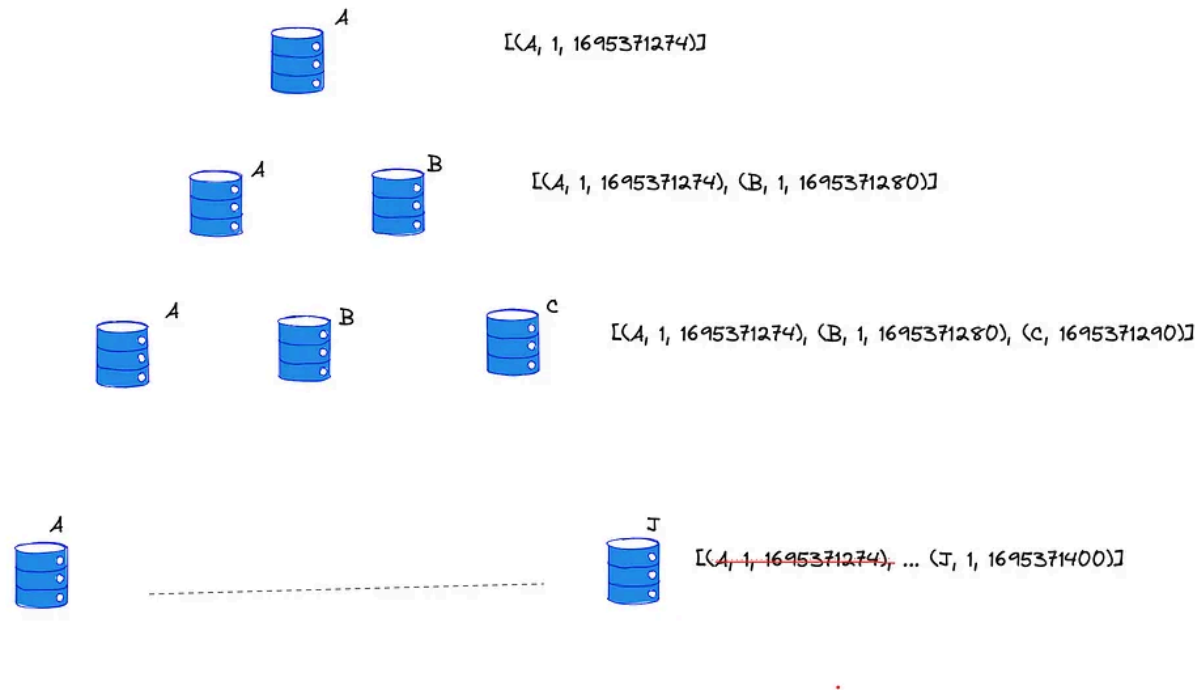

Fig 8.0: Clock truncation scheme

Fault tolerant

Handling temporary failures

Temporary failures includes— Data center failures happen due to power outages, cooling failures, network failures, and natural disasters.

Typically, distributed systems use a quorum-based approach to handle failures. A quorum is the minimum number of votes required for a distributed transaction to proceed with an operation. If a server is part of the consensus and is **down**, then we can't perform the required operation. It affects the availability and durability of our system.

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "**sloppy quorum**"; all the read and write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

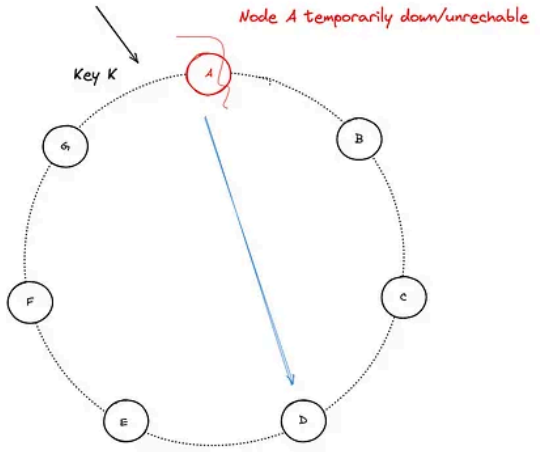

Fig 9.0: Hinted handoff example

Consider the example of Dynamo configuration given in above Fig 9.0 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a **replica that would normally have lived on A** will now be **sent to node D** with a **hint** in its metadata that suggests which node was the intended recipient of the replica. Nodes that receive hinted replicas will **keep them in a separate local database** that is **scanned periodically**. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, **D may delete the object** from its local store.

This approach is called a **hinted handoff**. Using it, we can ensure that reads and writes are fulfilled if a node faces temporary failure.

Handling permanent failures: Replica synchronisation

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. Dynamo implements an **Anti-entropy** (**replica synchronisation**) protocol to keep the replicas synchronised.

Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version.

To detect the inconsistencies between replicas faster and to minimise the amount of transferred data, Dynamo uses **Merkle trees.** A merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children.

If the hash values of the **root of two trees are equal**, then the values of the leaf nodes in the tree are equal and the **nodes require no synchronisation.**

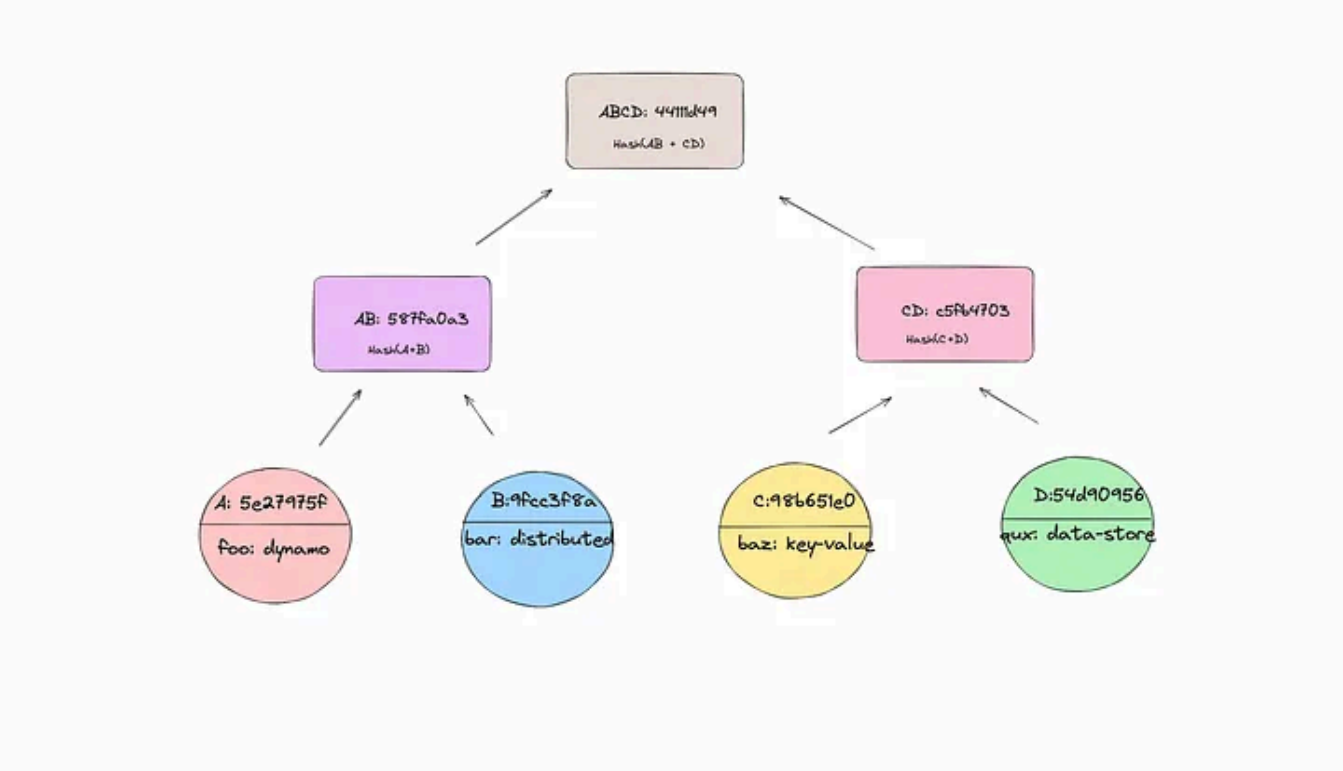

Fig 10.0: Merkle tree used for replica synchronisation

Advantage of Merkle tree:

- Each branch of the tree can be checked independently without requiring nodes to download the entire tree of the entire data set.
- Help in reducing the amount of data that needs to be transferred

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range it hosts.

Disadvantage of this scheme —Many key rangs change when a node joins or leaves the system thereby requiring the trees to be recalculated.

Membership and failure detection

The nodes can be offline for short periods, but they may also indefinitely go offline. We **shouldn't rebalance** partition assignments or fix unreachable replicas when a single node goes down because it's rarely a permanent departure. Therefore, the addition and removal of nodes from the ring should be done carefully.

Planned commissioning and decommissioning of nodes results in membership changes. These changes form history. They're **recorded persistently on the storage** for each node and **reconciled** among the ring members using a gossip protocol. A **gossip-based protocol** also maintains an **eventually consistent** view of membership. When **two nodes randomly choose one another as their peer, both nodes can efficiently synchronize their persisted membership histories**.

Let's learn how a gossip-based protocol works by considering the following example. Say node A*A* starts up for the first time, and it randomly adds nodes B*B* and E*E* to its token set. The token set has virtual nodes in the consistent hash space and maps nodes to their respective token sets. This information is stored locally on the disk space of the node.

Now, node *A* handles a request that results in a change, so it communicates this to *B* and *E*. Another node, *D*, has *C* and *E* in its token set. It makes a change and tells *C* and *E*. The other nodes do the same process. This way, every node eventually knows about every other node's information. It's an efficient way to share information asynchronously, and it doesn't take up a lot of bandwidth.

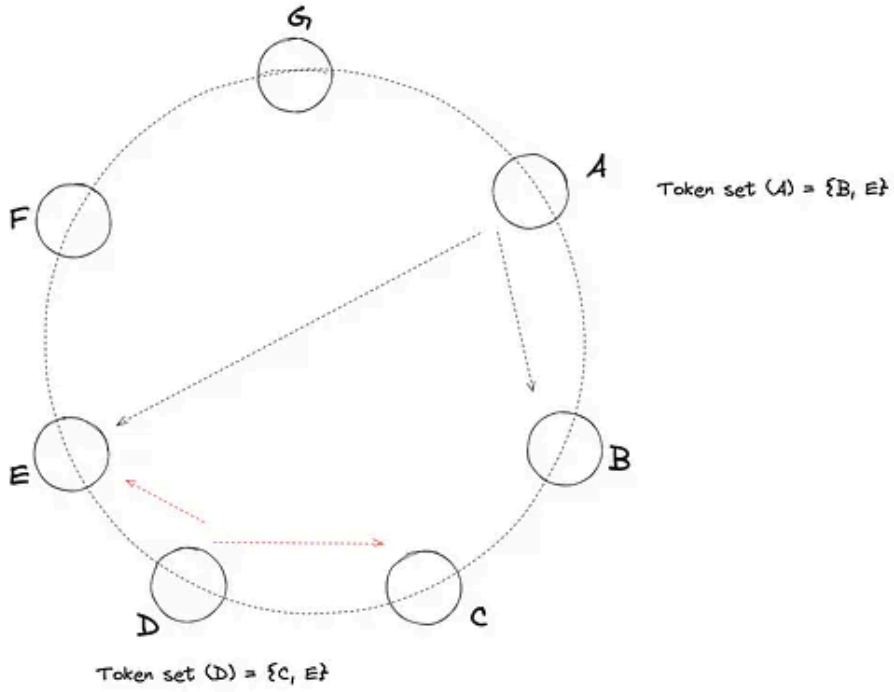

Fig 11.0: Gossip-protocol that enables each node learn about membership changes

Decentralised failure detection protocols use a gossip-based protocol that allows each node to learn about the addition or removal of other nodes. The join and leave methods of the explicit node notify the nodes about the permanent node additions and removals.

Dynamodb Scaling Software Architecture Software Architect Distributed Systems

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium